New Model Template

This project contains a template for creating a new neural model. This file documents the process which can be followed to update the template to then generate a working neural model.

This is split in to two sections, depending on if you just want to modify the differential equation of the core neuron, or if you also want to modify the synapse shaping, or number of synapse types, of the input.

Core Neuron Differential Equation

This template is set up to allow you to create your own core model differential equation, but use a provided synapse type (initially an exponentially decaying synapse type).

The template files to be modified are:

- c_models/src/neuron/models/neuron_model_my_impl.h
- python_models/neural_models/my_model_curr_exp.py
- c_models/src/neuron/models/neuron_model_my_impl.c
- c_models/src/neuron/builds/my_model_curr_exp/Makefile

In each of these files are a number of comments marked with "TODO"; these indicate the parts of the files that may need modification for your model, depending on what you are trying to achieve.

neuron_model_my_impl.h

This header file defines the data to be held for each neuron in a population. This includes the parameters of the neuron, as well as the current state; this might include the membrane voltage.

Any additional parameters or state variables should be defined between:
`typedef struct neuron_t {`
and
`} neuron_t;`

Parameters and variables should be 32-bit values (e.g. uint32_t, int32_t, REAL, UREAL, FRACT or UFRACT). This will ensure easy transfer of values between C and python code.

In addition, this file also defines the function `neuron_model_convert_input` This function can be used to scale the input, so long as it matches the python `weight_scale` parameter (see later).
The reason that this might be required is that the input is held in a 32-bit fixed point value as follows:

```
|1-bit sign|16-bit integer part|15-bit fractional part|
```

The smallest number that can be represented in this form is ~0.000030517578125. If the input is to be shaped (e.g. there might be an exponential decay of the input in some neural models), this precision might not be enough for the shaping to be accurate. In this case, if small fractional inputs are expected in general, some scaling might make sense.

my_model_curr_exp.py

This file is used to allow the model to be used in a PyNN script. This takes the parameters defined in the PyNN script and translates them into a form that can be used by the C code. It is very important therefore that this file matches the C code precisely.

The first thing to decide here is if the model is to make use of any standard components provided by sPyNNaker. If so, you can add these to the class as indicated. In this example, the model makes use of the exponential synapse shaping, and so the model class inherits from `AbstractExponentialPopulationVertex` .

The next thing to define is the `CORE_APP_IDENTIFIER` . This is an arbitrary identifier that helps to ensure that the correct data is loaded by the correct application binary. It isn't essential that it is unique, although this is preferred if possible. This will also be mirrored in the `Makefile` (see later).

The `_model_based_max_atoms_per_core` is an absolute upper limit on the number of neurons that can be run on any core of the SpiNNaker machine. At present, some of the data structures limit this to 256 as an absolute limit. If your model is particularly complex, you might have to reduce this number; unfortunately, only experimentation and estimation can help with this at present. If you are unsure, leaving this value at 256 should be fine, especially for testing single neuron networks.

The parameters of the model that can be defined in pyNN will be in part dictated by the extra components inherited from, as defined above. In this example, the `AbstractExponentialPopulationVertex` defines `tau_syn_E` and `tau_syn_I` , so these are added as parameters here to be passed on to that component. The remaining additional parameters are likely to be values for the variables defined in the `neuron_t` structure in `neuron_model_my_impl.h` , as specified above.

Once the parameters have been defined, the `n_params` value must be set to the number of parameters in the `neuron_t` structure defined above. Note that synapse parameters (such as `tau_syn_E` and `tau_syn_I`) are *not* included in this count.

The value of `binary` should match the name of the binary to be created by compilation of the C code. This must match the value in the `Makefile` (see later).

The `weight_scale` value is used to scale inputs as described above. This scaling must be reversed by the `neuron_model_convert_input` function in `neuron_model_my_impl.h` e.g. if the scaling here is 1024.0, the inputs will be multiplied by 1024.0, and so must be divided by 1024.0 in `neuron_model_convert_input` . As the ARM core doesn't support divide well, it is recommended that the scaling is a power of 2; this allows the divide to work with shift instructions.

Any classes inherited from have to be instantiated, and have their parameters passed in. In the example, the `AbstractExponentialPopulationVertex` is initialized with its required parameters.

The parameters passed in to the class not passed on to a component module, must be stored so that they can be used later on in the class.

The `model_name` function returns the name of the model. This is not critical, but is used in reports.

The sPyNNaker system attempts to place neurons on a core depending on the total amount of CPU cycles that might be used in the processing of those neurons. To this end, an estimate of how many CPU cycles are used by a range of atoms is returned by the `get_cpu_usage_for_atoms` function. Like the `_model_based_max_atoms_per_core` above this is currently likely to be an estimate, and won't be critical in single neuron experiments.

The `get_parameters` method returns the parameters that will be passed to the C code. The number of, order of and type of these parameters is critical - they must match those defined in the `neuron_t` structure in `neuron_model_my_impl.h` . The data types map as follows:

|C data type|Python data type| |REAL|DataType.S1615| |UREAL|DataType.U1616| |FRACT|DataType.S031| |UFRACT|DataType.U032| |uint32_t|DataType.UINT32| |int32_t|DataType.INT32|

Certain inherited components will require specific functions to be defined to work correctly. These are marked with `@abstractmethod` in the component source code. For example, the `AbstractExponentialPopulationVertex` requires the function `is_exp_vertex` to be defined.

neuron_model_my_impl.c

This file defines the details of how to update the neuron state at every time step. This will use the parameters defined in `neuron_model_my_impl.h` , along with excitatory and inhibitory input to update the state variables defined in the same place.

The main state update takes place in `neuron_model_state_update` . Here, the overall input is calculated and then can be used to update the neuron state. Finally, this function must determine if the neuron has spiked; if so, it should return `true` otherwise `false` should be returned. In the example, the current is simply added forever, and the model never results in a spike. Clearly this is not very useful in a neuron model.

The `neuron_model_get_membrane_voltage` function can be customized to get the membrane voltage from the neuron data structures. If the membrane voltage is not explicitly stored, it can be calculated here.

The `neuron_model_print` function is used for debugging. This should print out the information from the defined neuron structure that is deemed to be useful during debugging.

Makefile

The Makefile is used to build the C code. It contains a number of parameters that must match those defined in the Python code. There are a number of variables that are defined outside of this Makefile which are useful:

- `EXTRA_SRC_DIR` : This points to the `c_models/src` directory of this project. Any source files in this project can be referenced using this variable as a starting point.

- `SOURCE_DIRS` : This points to the `neural_modelling/src` directory of the sPyNNaker source code. Any pre-existing component source files from sPyNNaker can be referenced using this variable as a starting point.

`APP` must match the `binary` parameter in `my_model_curr_exp.py` , but without the `.aplx` extension.

`MODEL_OBJS` lists the sources to be included in the build, with the `.c` extension replaced by the `.o` extension. In this case, only the neuron model is included, along with the synaptic plasticity rules (in this case it uses the static implementation, which means no synaptic plasticity is defined); this could be extended if the model is complex enough to require the code to be defined in multiple .c source files.

`NEURON_MODEL_H` is set to the header file of the neuron model created above (i.e. `neuron_model_my_impl.h` in this example).

`SYNAPSE_TYPE_H` is set to the header file of the synapse shaping to be used. In this example, as the Python code is using `AbstractExponentialPopulationVertex` , the c code makes use of the corresponding `synapse_type_exponential_impl.h` .

`APPLICATION_MAGIC_NUMBER` must match the `CORE_APP_IDENTIFIER` from `my_model_curr_exp.py` . This will be used to verify that the data being read is meant for this application.

New Synapse Type

If a new synapse shaping or number of synapse types is required, additional files must also be modified. The files involved in this example are:

- c_models/src/neuron/synapse_types/synapse_types_my_impl.h
- python_models/neural_models/my_synapse_type.py
- python_models/neural_models/my_model_curr_my_synapse_type.py
- c_models/src/neuron/builds/my_model_curr_my_synapse_type/Makefile

As with the example above, each of these files are a number of comments marked with "TODO" to indicate the parts of the files that may need modification for your model, depending on what you are trying to achieve.

synapse_types_my_impl.h

In contrast to the neuron models, the synapse types are implemented entirely in the header file. This is for reasons of efficiency.

The first thing to be determined is how many synapse types are to be allowed. In the case of a single excitatory and inhibitory synapse for each neuron, there are two types, but any number of types can be supported; for example, a dual exponential synapse model has been created with two different excitatory synapses supporting different time constants. Once this has been decided, `SYNAPSE_TYPE_COUNT` can be set to this number, with `SYNAPSE_TYPE_BITS` representing the number of binary digits required to represent this number e.g. 1 bit can represent 1 or 2 synapse types and 2 bits can represent any number up to 4 synapse types.

Following this, the actual parameters for each of the synapses can be decided. The same parameters are expected to be provided for each synapse type for each neuron (although with different values). The parameters can be filled in between

```
typedef struct synapse_param_t {
```

and

```
} synapse_param_t;
```

As with the neuron model types, 32-bit types should be used for each of the parameters.

The `synapse_types_shape_input` function should take the input buffers and transform the buffer for the given neuron with the given synapse parameters, depending on what is to be acheived by the synapse shaping.

The `synapse_types_add_neuron_input` function should take the input buffers and add to the buffer for the given neuron and synapse index, the given input, using the given synapse parameters to compute the initial value to add. This allows the synapse type to do any required transformation on the initial input value, e.g. a synapse that decays over a number of timesteps would not normally add the entire input weight at the start, but rather spread it over the decay curve.

The `synapse_types_get_excitatory_input` function should return the total excitatory input from the synapses of a given neuron. If there is only one excitatory synapse per neuron, it can return the single value of this input, but if there is more than one, it might return the sum of these inputs, or perform a more complex calculation.

Similarly, the `synapse_types_get_inhibitory_input` function should return the total inhibitory input from the synapses of the given neuron. There can be a different number of excitatory and inhibitory synapses.

The `synapse_types_get_type_char` and `synapse_types_print_input` are only ever used for debugging purposes. The former should return a string representing the given neuron type, and the latter should print the inputs.

my_synapse_type.py

This file is the python counterpart to `synapse_types_my_impl.h` and so certain parts of these files must match.

The parameters defined in the header file can be passed in here. A single value can be used for all the synapse types, or a different parameter can be provided for each synapse type; in the example one is provided for each of the excitatory and inhibitory types. These parameters should be stored for use in the other functions.

The `get_n_synapse_parameters_per_synapse_type` function must return the number of parameters defined in `synapse_param_t` in the `synapse_types_my_impl.h` header file.

The `get_n_synapse_types` function must return the same value as `SYNAPSE_TYPE_COUNT` is set to in `synapse_types_my_impl.h` .

The `get_n_synapse_type_bits` function must return the same value as `SYNAPSE_TYPE_BITS` is set to in `synapse_types_my_impl.h` .

`write_synapse_parameters` must write the synapse parameters per neuron, in the order that they are defined in the header file.

my_model_curr_my_synapse_type.py

This file should be completed in a similar way to `my_model_curr_exp.py` described above. The main difference is that the inheritance is different, and so the required parameters and inheritance initialization is different.

Note also that the `binary` parameter is different, as is the `CORE_APP_IDENTIFIER` . These will have to be reflected in the Makefile (see later).

Makefile

The editing of the Makefile for this part is similar to the Makefile described above. The main differences are the `APP` name, which should match the `binary` parameter in the Python and the `APPLICATION_MAGIC_NUMBER` which should match the `CORE_APP_IDENTIFIER` . The `SYNAPSE_TYPE_H` in this Makefile is set to the new synapse type header described above.

Of note is that the `MODEL_OBJS` is the same in both Makefiles. This shows how the neuron model can be reused with different synapse types once created. Similarly, different STDP rules can be added in to the neuron models, but this is outside of the scope of this document.

Compilation and execution of the models

Model compilation

Once the Makefile for the model has been written, compilation simply consists of running `make` in the directory containing the Makefile. Assuming there are no compilation issues, this will result in a binary with a ".aplx" extension being created in the `python_models/model_binaries` subfolder.

Binary search path

The `model_binaries` folder is added to the sPyNNaker binary search path by the code found in `python_models/__init__.py` . Should a different path be required, it can be added in the same way.

Model execution

The script `examples/my_example.py` shows how you can use your new model. This is a standard pyNN script, except that the new python models are also imported into the script at the top. Once imported, the models are used in the same way as any other pyNN model; the model parameters that can be specified are those that were added in the examples above.

The example script shown here runs a single neuron of each of the new models, and then plots a graph of the membrane potential against the simulation time.

A note on Makefile.common

In `c_common/neuron/builds` there is a file called Makefile.common. This Makefile is included by each Makefile in the builds subfolders. This defines a few additional variables, which can be used to add additional STDP components, namely `EXTRA_STDP` , `EXTRA_STDP_WEIGHT_DEPENDENCE` and `EXTRA_STDP_TIMING_DEPENDENCE` . Any additional components added here just indicate the existence of the component and set up appropriate make rules for these components. To actually make use of the components, they must also be added to the `MODEL_OBJS` variable in the final Makefile of the binary.